ME5418: Machine Learning in Robotics

Quadruped Robot Locomotion with Reinforcement Learning

Final Report

Name: Ho Kae Boon

Student Number: A0219811H

23th November 2023

1 Introduction

1.1 Background

Traditional wheel and track robots are often limited to a well-structured environment such as roads. In comparison, legged robots, such as quadrupeds that are inspired by four-limbed mammals like dogs, offer greater versatility. Their ability to navigate complex and unstructured terrains, including forests, caves, stairs, and rubble, makes them well-suited for applications such as search and rescue, exploration, transportation, and surveillance across diverse landscapes.

A notable real-world application was during the 2020 pandemic when the Singapore Government employed the Boston Dynamics' Spot [1] quadruped robot to monitor Ang Mo Kio Park for safe distancing measures [2]. Spot's success in traversing the park's complex terrain has highlighted the significance of the engineers' work in designing effective locomotion algorithms for the robot. As such, the designing of quadruped locomotion strategies is crucial to ensuring these robots move efficiently and adaptably across various terrains, contributing to their effectiveness in real-world scenarios.

1.2 Existing conventional solutions

The simplest conventional solution for quadruped locomotion involves using a Central Pattern Generator (CPG) to generate rhythmic gait patterns [3], such as trotting or galloping. The advantage of this approach is its ease of implementation and stability in a predictable environment, like a planar ground. However, due to its open-loop control nature, it falls short when faced with environmental variations, such as uneven terrain.

Another widely used conventional approach for quadruped locomotion is Model Predictive Control (MPC) [4]. MPC relies on an accurate model of the robot to generate a joint trajectory with feedback correction. While it performs well on various uneven terrains, its drawback is the complexity of implementation, mainly due to its dependence on an accurate kinematic and dynamic model. Engineers must calculate inverse kinematics and equations of motion by establishing relationships between joint angles, positions, velocities, masses, inertia, and torque variables for each link of the robot. This process may involve substantial mathematical derivations and computations, and thus can be challenging. I have provided video to both CPG and MPC, titled 'HoKaeBoon_Conventional_Policies.mp4".

In contrast to these approaches, the advantages of employing deep reinforcement learning methods, such as Deep Deterministic Policy Gradient (DDPG) [5] to teach quadruped locomotion become clear. Reinforcement learning does not require accurate modelling of the robot's kinematic and dynamic information. It also allows for an adaptive gait policy to traverse in an uneven terrain. However, a downside of using reinforcement learning for obtaining quadruped locomotion policy is the potential wear and tear on the physical robot if it explores policies on its own. Additionally, when trained in a digital double, the policy may not seamlessly translate well into the actual robot. Compared to other deep reinforcement learning methods such as Proximal Policy Optimization (PPO) [6] and Deep Q Network (DQN) [7], DDPG is particularly well-suited for continuous state and action space usage. Therefore, it is better suited for tasks involving quadruped locomotion learning. This is the primary reason we chose to use DDPG in our project.

1.3 Project’s problem statement

The goal of the project is to use reinforcement learning, DDPG in particular, to task any quadruped robot with developing an adaptive gait policy, enabling it to move towards the front direction in any given environment.

For this project, we have chosen the AlienGo quadruped robot from UniTree [8]. The robot will be equipped with IMU sensors and force torque sensors to obtain proprioceptive information, such as the spatial position, orientation and velocities of the robot's torso, as well as the detection of leg contact with the ground. No external perception sensors is employed.

The robot has 16 joints that operate on a velocity-controlled high-torque motor. In this project, we will be utilizing only the upper limb joints and lower limb joints, totalling 8 joints, while freezing the other joints to simplify our problem. The input to these motors is the continuous torque values.

In this project, the environment we will be using includes only a planar ground environment and an environment with zero gravity. We got to explore only these two environments due to project complexity and time limitations. Our environment is single-agent, static, deterministic, and partially observable by our robot.

2 Implementation

2.1 RL cast

The formulation of our problem statement within the reinforcement learning framework differs largely from our initial proposal. Thus, I will redescribe our state space, action space and reward structure again here.

Our state space comprises 34 values, including the following:

- Position of the robot torso's center of mass (3 values: x, y, and z-axis position)
- Velocity of the robot torso's center of mass (3 values: x, y, and z-axis velocity)
- Orientation of the robot torso's center of mass (4 values: 3D rotational angle in quaternion format)
- Rotation velocity of the robot torso's center of mass (3 values: 3D rotation velocity in Euler format)
- Angle of the robot joints (8 values: 1D rotation angle for each joint)
- Rotation velocity of the robot joints (8 values: 1D rotation velocity for each joint)
- Ground contact of robot legs (4 values: binary truth values for each leg)
- Robot termination state indicator (1 value: binary truth value)

All spatial values are obtained with respect to the world frame of our environment, as shown in Figure 1.

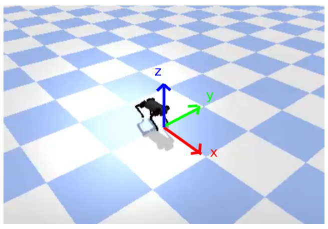

Figure 1: World Frame in PyBullet Environment

We have implemented two distinct definitions for termination states. In Version 1, termination occurs under any of the following conditions, while Version 2 considers only the third condition:

• Collision between any two robot links.

• Robot torso's height (z-position) falling below 0.2 m

• Robot torso's lateral position (y-position) exceeding ± 1.5 m.

The action space comprises 8 continuous values, corresponding to the joint torque values of the robot.

We have implemented two distinct reward structures. Reward Structure Version 1 is designed with the intention of encouraging the robot to maintain a forward velocity, a torso height of 0.45m, and a stable torso orientation when walking. Additionally, it aims to minimize joint movement and avoid termination states for as long as possible. The structure is as follows

$$r_t = 5v_x - 20(z - 0.45)^2 - 20\theta^2 - 0.02\sum_{i=1}^{8} u_i^2 + r_{time}$$

where:

• v_x is the torso's x-velocity,

• z is the torso's height,

• θ is the torso's pitch angle,

• u_i is the action value for joint i,

• $r_{time} = \{ \begin{array}{l} 10\frac{t-15}{15}\ when\ 15 < t \leq 30; \\ 15\frac{t-30}{20}\ when\ 30 < t \leq 50; \\ 25\ when\ t \geq 50 \end{array} \}$ with t as the timestep

Reward Structure Version 2 is designed to be more simple, focusing solely on the reward for forward velocity and a penalty for lateral position. The structure is as follows:

$$r_t = 5v_x - 1.5y^2$$

where v_x is the torso's x-velocity, and y is the torso's y-position.

The rationale behind employing two different termination state definitions and reward structures will be further elaborated upon in the Results section below.

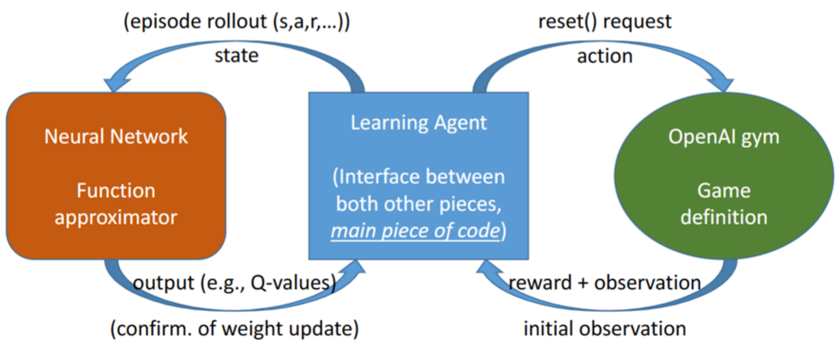

Figure 2: Relationship between Gym, Neural Network, and Learning Agent (from Lecture slide) [9]

Our RL framework implementation consists of three components: OpenAI Gym, Neural Network, and Learning Agent. The code for each part can be found in 'aliengo_gym.py,' 'DDPGNet.py,' and 'main_code.ipynb and .py,' respectively. The OpenAI Gym part defines the environment with which our agent interacts. The neural network part defines the structures for the actor, critic, and target networks, along with their forward and backward passes. The Learning Agent is the main driver code, and serves as the middle bridge between OpenAI Gym and Neural Network, as shown in Figure 2 above.

2.2 OpenAI Gym

We have defined an 'AliengoEnv()' class that inherits from the OpenAI 'gym.Env' class. You may refer to the second paragraph of our previous OpenAI report for detailed information on the initialization of the 'AliengoEnv()' class (previously named as 'SAPPEnv()' class).

In our implementation, we have introduced several methods to provide instructions and extract information from the simulation, you can refer to our OpenAI report for details of their implementation. However, a few changes have been made to each of these methods, as follows:

- reset(): This method resets the robot to its starting position and orientation. In our final submission, we have added an extra option to toggle gravity on or off. This is achieved by providing an additional argument, 'Policy' (corresponding to our trained policy index), during the initialization of our environment class. When the 'Policy' argument is set to 3, gravity is turned off.
- get_state(): This method returns the current state information, expanded to include the 34 values as detailed in the RL cast section. The checking of state termination is done via the is_terminate() method.

- get_observation(): This method returns the agent's observation, including every state variable except for the 'is_terminate' variable.
- is_terminate(): This is a new method that returns a boolean value indicating whether the current state is a termination state. The choice of termination definition depends on the 'Policy' value provided during class initialization.
- step(action): This method prompts the Aliengo robot to execute a timestep of joint rotation for all eight joints. It returns the following: 1) previous_state; 2) action taken; 3) new state; 4) reward; 5) is terminate; 6) previous action taken. The reward value is calculated by calling the 'get_reward()' method, with the choice of reward structure depending on the 'Policy' values.
- There are no changes to the 'render()' and 'disconnect()' methods from our previous OpenAI report explanation.

2.3 Neural Network

For detailed information on the definition of the actor, critic, and target networks, as well as their forward and backward passes, please refer back to our Neural Network report. In our final submission, we made a few small changes distinct from the report:

- Input Dimension Change: The dimensions of the observation and next observation input placeholders are modified to (None, 40, 1). This adjustment is due to our new observation space, which consists of 32 values (as explained earlier). Additionally, we append the 8 previous action values to the observation input, thus resulting in a total of 40 values.
- Normalization Approach: The 'scale_action()' and 'scale_observation()' functions now employ a normalization approach with the following formula: $x_{new} = \frac{x_{old} - x_{old\ min}}{x_{old\ max} - x_{old\ min}} (1 - (-1)) + (-1)$.
- Regularization and Gradient Clipping: During the training process of both the Actor and Critic networks, we have introduced L2 regularization to prevent overfitting. Additionally, gradient clipping is applied to address issues related to exploding gradients

2.4 Learning Agent and training process

For a detailed explanation of our Learning Agent implementation as well as the training processes, please refer back to our Learning Agent Report. There are minimal changes to the Learning Agent compared to our previous submission. The only modifications are as follows:

- Script File Renaming: The script file is now named "main_code.ipynb"
- Python file: We have also written a Python version of the script named “main_code.py”
- OUNoise() Class Adjustment: Within the 'DDPGNet.py' file, under the 'OUNoise()' class, the decay of the degree of perturbation is simplified by using only one parameter, 'decay_rate.' The decay equation is now expressed as follows: *self.sigma *= self.decay_rate*.

3 Results

We have trained multiple policies, making adjustments to the parameters along the way. In this final submission, we would like to highlight three specific trained policies referred to as Policy 1, 2, and 3. For a visual representation of each policy, please refer to our video titled 'Trained_Policy_Video.mp4.

3.1 Policy 1

For our first policy, the goal was to train the robot to walk smoothly and stably on a planar surface, maintaining an upright posture with minimal joint movements. Therefore, we utilized termination definition V1 and reward structure V1.

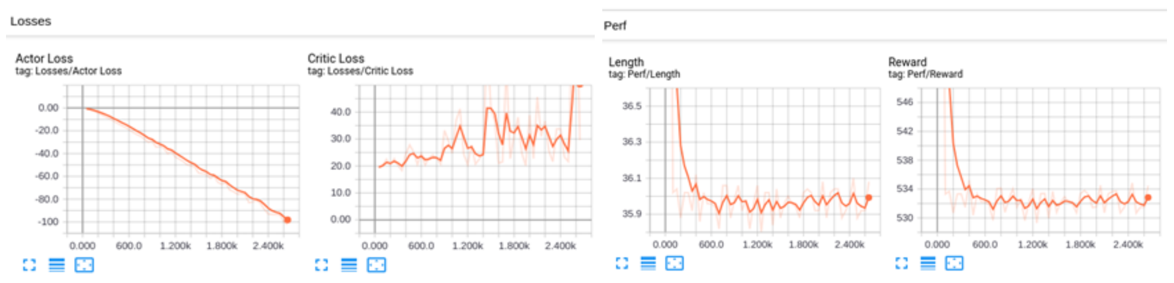

Figure 3: Graph of losses, episode length and episode reward over episodes via Tensorboard during training of Policy 1

Ideally, a successful policy should result in an increasing episode reward over time. However, as shown in Figure 3 above, the cumulative episode reward remains consistently low throughout the training episodes. This could be attributed to the robot getting stuck at a local suboptimal policy. In fact, the rendered video reveals that in an attempt to maximize the forward velocity reward, the robot often chooses to fall forward until it touches the ground, leading to a quick termination of the episodes with a low overall reward.

Despite our efforts to refine the reward policy and introduce noise for increased exploration, achieving a satisfactory policy proved challenging even after hours of training. We recognized that exploration through noise takes time, and restricting ground contact only added complexity to the problem. Consequently, we opted to simplify the task by adjusting our termination definition and reward structure. This involved removing the prohibition on touching the ground and eliminating penalties for torso instability and large joint movements. This led to the development of Policy 2.

3.2 Policy 2

We trained Policy 2 using termination definition V2 and reward structure V2.

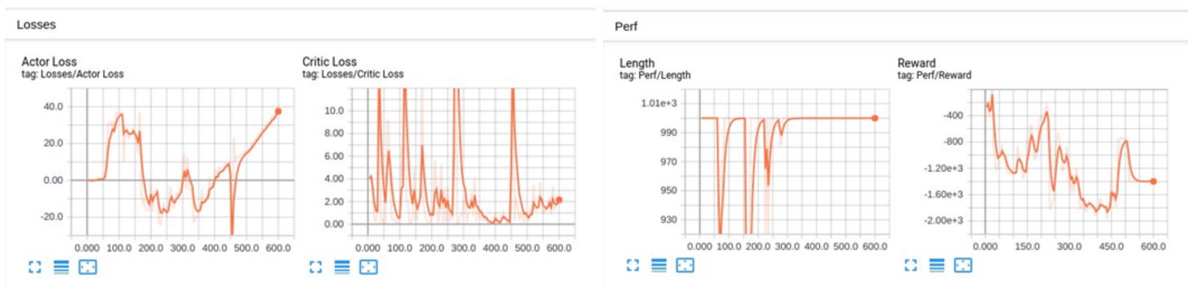

Figure 4: Graph of losses, episode length and episode reward over episodes via Tensorboard during training of Policy 2

By removing the termination condition related to the robot touching the ground, we allowed the agent more freedom to devise strategies for forward movement even after falling down. Throughout the training, we can notice that Figure 4 above shows more variation in reward values. Notably, two reward peaks are evident around 200 episodes and 500 episodes, representing local maximum suboptimal policies. At the 500th episode of the trained policy, the quadruped successfully crawls in the x-direction. Although it eventually deviates from the x-axis direction, this achievement represents the best policy we could obtain within hours of training.

3.3 Policy 3

As a complement to Policy 2, which operates in a standard ground environment, we sought to explore a different scenario. To do this, we employed a zero-gravity PyBullet environment, simulating a scenario where the robot is placed in space and needs to teach itself to propel forward. The motivation behind this is to showcase the adaptability of our reinforcement learning framework for the robot to learn new locomotion strategies in an all-new environment. After approximately 200 episodes of training, we observed the robot is able to propel itself forward, albeit at a very slow and inefficient pace.

4 Discussion

4.1 Advantages and Limitations of our approach

Our implementation of DDPG offers several advantages. This includes convenient monitoring of training performance using Tensorboard and the generation of saved GIFs. Furthermore, the model can be effortlessly saved and loaded, allowing for training to be conducted across multiple runs. Credit for these benefits goes to our professor [9], as we largely imitate his lecture's code structure.

The limitation of our approach mainly lies in training inefficiency. We use a single quadruped in a PyBullet environment, trained solely with a CPU. The absence of parallel training and GPU usage results in a prolonged training time required to reach an optimal policy. We came across similar projects online that implemented parallel training with thousands of quadrupeds [10]. On planar terrain, the optimal policy can be obtained within a few minutes, while on uneven terrains, it can take up to 20 minutes [10]. With our current implementation, we lack both the capacity and time to enable the quadruped to continuously explore improved policies.

4.2 Comparison with state-of-the-art conventional alternative

Unfortunately, our trained model did not meet the expected standards, as it struggled to learn even the basic gaits of trotting or galloping. The performance was comparatively inferior to even the traditional CPG approach. However, it is essential to note this situation is only specific to our implementation, and it should not discredit the advantages of using the DDPG reinforcement learning approach to tackle the quadruped locomotion problem. Thus, despite these setbacks, I would still like to draw comparisons between our ideas with one of the state-of-the-art alternative methods which is perceptive locomotion with MPC [11].

All of the locomotion strategies discussed in this paper operate without relying on visual input information. Hence, they exhibit a lower degree of adaptability to the environment when compared to state-of-the-art locomotion strategies that leverage perception data, typically obtained through depth cameras or LiDAR. These advanced strategies capture geometric information from the surroundings. Through techniques like plane segmentation, the robot can identify steppable regions from unsteppable ones. Utilizing MPC, the robot can then plan a trajectory for foot placement in these regions, resulting in smooth and adaptive locomotion. The only downside to perception locomotion is that it still relies on a model-based approach, thus requiring a complex dynamics and kinematics analysis. I have provided video to this policy inside the "HoKaeBoon_Conventional_Policies.mp4".

4.3 Challenges encountered during implementation

This project is more challenging than we initially assumed during the proposal. Our problem possesses a high-dimensional and continuous state space and action spaces. This results in a need for extensive exploration, that has to be carefully balanced with exploitation to obtain an effective policy. This can only be done by tuning the many hyperparameters which are very sensitive and will largely impact the learning trajectory. Moreover, it is not easy to design a perfect reward structure that strikes a balance between various conditions. Another challenge is due to the lack of parallel training with multiple robots which largely increases the time-consumption needed for training.

4.4 Lesson learned

Attempting to overcome the array of challenges throughout this project has been an invaluable learning experience for me. I have gained a deeper insight and appreciation of the complexity of applying reinforcement learning to robot locomotion tasks, especially the various factors that have to be taken into consideration. Working hands-on with this project allowed me to put my theoretical knowledge learned from lecture videos into practical use. All in all, this project has largely expanded my technical skills and prepared me to explore further the field of machine learning in robotics.

4.5 Potential future work

To further improve our current project, we can consider adding additional layers to our neural network design to allow our robot agent to learn more complex policies. It is also crucial to explore parallel training using software like Isaac Gym [12] which is more optimized than Pybullet for such applications. Ongoing adjustments to our termination state definition and reward structure, along with meticulous tuning of each hyperparameter, are also essential for improving the efficiency of the learning process. Once we are able to obtain a good policy in an even ground stable environment, we can consider progressing to a more complicated environment such as uneven terrains. Furthermore, we can also explore training in a dynamic environment by introducing elements such as random obstacles or randomly generated terrains.

5 Acknowledgement

When developing our code, we used Professor Guillaume's lecture demo code as a starting point and for further modification. When working with the PyBullet library [13], Tensorflow library [14], and OpenAI's gym library [15], we turned to their sample codes and official documentation for insights on implementing various functions. For the neural network, we drew inspiration from the existing DDPG network code on GitHub [16]. However, due to the difference in our Tensorflow version, we still mainly code that segment from scratch. Finally, we would like to offer our gratitude to the professor [9] and the TA for their timely feedback and guidance throughout the projects.

6 References

[1] "Spot | Boston Dynamics," *Boston Dynamics*. https://bostondynamics.com/products/spot/ (accessed Nov. 24, 2023).

[2] C. Tan, "Robot reminds visitors of safe distancing measures in Bishan-Ang Mo Kio Park," *The Straits Times*, May 09, 2020. Accessed: Nov. 24, 2023. [Online]. Available: https://www.straitstimes.com/singapore/robot-reminds-visitors-about-safe-distancing-measures-in-bishan-ang-mo-kio-park

[3] X. Zhang, Z. Xiao, Q. Zhang, and W. Ping, "SYNLOCO: Synthesizing Central Pattern Generator and Reinforcement Learning for quadruped locomotion," *arXiv (Cornell University)*, Oct. 2023, doi: 10.48550/arxiv.2310.06606.

[4] Z. Zhang, X. Chang, H. Ma, H. An, and L. Lin, "Model predictive control of quadruped robot based on reinforcement learning," *Applied Sciences*, vol. 13, no. 1, p. 154, Dec. 2022, doi: 10.3390/app13010154.

[5] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *arXiv (Cornell University)*, Sep. 2015, [Online]. Available: http://export.arxiv.org/pdf/1509.02971

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy optimization Algorithms," *arXiv (Cornell University)*, Jul. 2017, [Online]. Available: http://export.arxiv.org/pdf/1707.06347

[7] V. Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning," *arXiv (Cornell University)*, Dec. 2013, [Online]. Available: http://cs.nyu.edu/~koray/publis/mnih-atari-2013.pdf

[8] “Aliengo,” *UnitreeRobotics*. https://shop.unitree.com/products/aliengo (accessed Nov. 24, 2023).

[9] “MARMoT Laboratory @ NUS-ME: Home.” https://www.marmotlab.org/ (accessed Nov. 24, 2023).

[10] N. Rudin, D. Hoeller, P. Reist, and M. Hutter, “Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning,” *arXiv*, Sep. 2021.

[11] R. Grandia, F. Jenelten, S. Yang, F. Farshidian, and M. Hutter, “Perceptive locomotion through Nonlinear Model-Predictive Control,” *IEEE Transactions on Robotics*, vol. 39, no. 5, pp. 3402–3421, Oct. 2023, doi: 10.1109/tro.2023.3275384.

[12] Nvidia-Omniverse, “GitHub - NVIDIA-Omniverse/ISAACGymENVS: ISAAC Gym Reinforcement Learning Environments,” *GitHub*. https://github.com/NVIDIA-Omniverse/IsaacGymEnvs (accessed Nov. 24, 2023).

[13] “GitHub - bulletphysics/bullet3: Bullet Physics SDK: real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning etc.,” *GitHub*. https://github.com/bulletphysics/bullet3 (accessed Nov. 24, 2023).

[14] “tensorflow,” *GitHub*. https://github.com/tensorflow (accessed Nov. 24, 2023).

[15] Openai, “GitHub - openai/gym: A toolkit for developing and comparing reinforcement learning algorithms.,” *GitHub*. https://github.com/openai/gym (accessed Nov. 24, 2023).

[16] Cookbenjamin, “GitHub - cookbenjamin/DDPG: Clean Python Implementation of the Deep Deterministic Policy Gradients Algorithm,” *GitHub*. https://github.com/cookbenjamin/DDPG (accessed Nov. 24, 2023).